

EE49 Laboratory 4 Demodulation

Due Date: 5/20/2011

After completing this lab assignment, you must be checked off by a TA in lab and provide your code via email. If you cannot make the allotted timeslots, it is your responsibility to setup an alternate appointment with the TAs before the due date. Good luck!

GOAL: The goal of this lab is to learn about demodulation, and implement the portions of a wireless BPSK receiver for a USRP-to-USRP link. Specifically we will be implementing a simple channel correction scheme, a BPSK symbol de-mapper, a preamble detection scheme, packet decoding, and a CRC error check.

Note: Most of the control traffic (especially LabView error handling) has been done for you, and a few subVIs will be provided which should be helpful.

1 Primer

Here's the basic lab setup: Similar to the previous lab, you will be given a TX and an RX code. This time however, the transmitter is designed for you. Infact, it is quite similar to the one you designed in the previous lab with the only difference being the bit generation (a meaningful message will be transmitted instead of a random bit generator). We'll tell you the basic USRP2 parameters to use (carrier frequency, samples/bit, etc.). Your job is to finish the RX code, i.e. receive and decode the transmitted message.

(1) Transmitter Packet Format



Figure 1: Tx Packet Format

The transmitting lab USRP is sending a stream of packets in the format shown above. There are:

- 31 bits of a pseudo-noise (PN) code preamble/sync (whose purpose we will come to shortly!). We will provide you with the value of these 31 bits.
- followed by 30 bytes(240 bits) of data in ASCII format
- followed by 1 byte(8 bits) of CRC error checking.

(2) High-level Receiver Block Diagram

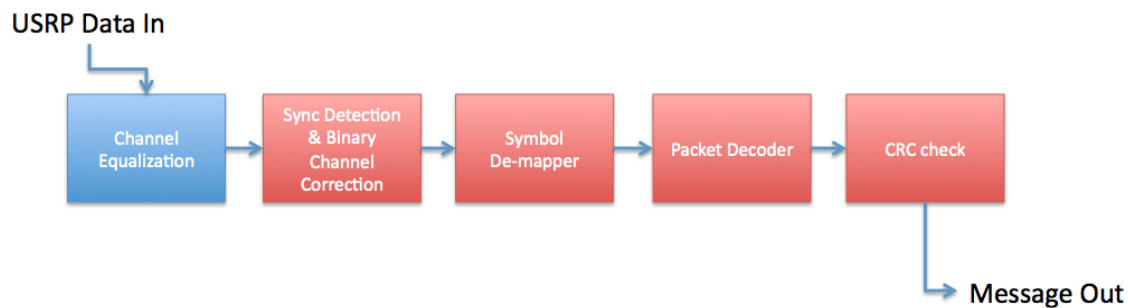


Figure 2: Receiver Block Diagram

At a high level, the receiver block diagram is shown in figure 2 above. The block in blue is already completed for you. The blocks in red are not complete, and are up to you to get working. At the end of this lab, you'll have a simplified but functional wireless receiver! We briefly review each block in this primer, and provide more detail in the following sections.

(3) Channel Equalization

As aforementioned, this block is completed for you. While it is a straightforward idea, channel equalization is quite difficult to implement in practice. **You are not responsible for designing this block, but it is imperative that you understand the basic idea behind it.**

Here's the basic idea: We've implemented a "**blind**" equalization scheme, which means it corrects the phase and frequency offset of the channel without knowing *anything* about the transmitted message except for the modulation scheme (i.e. BPSK). Sounds great, but it turns out there is one problem...

For BPSK, the equalizer expects that the incoming symbols should have one of the two values sent by the transmitter, i.e. $+j$ or $-j$ as in lab three (see figure 3 for a refresher on the BPSK symbol map). Essentially, the blind equalizer looks at a large

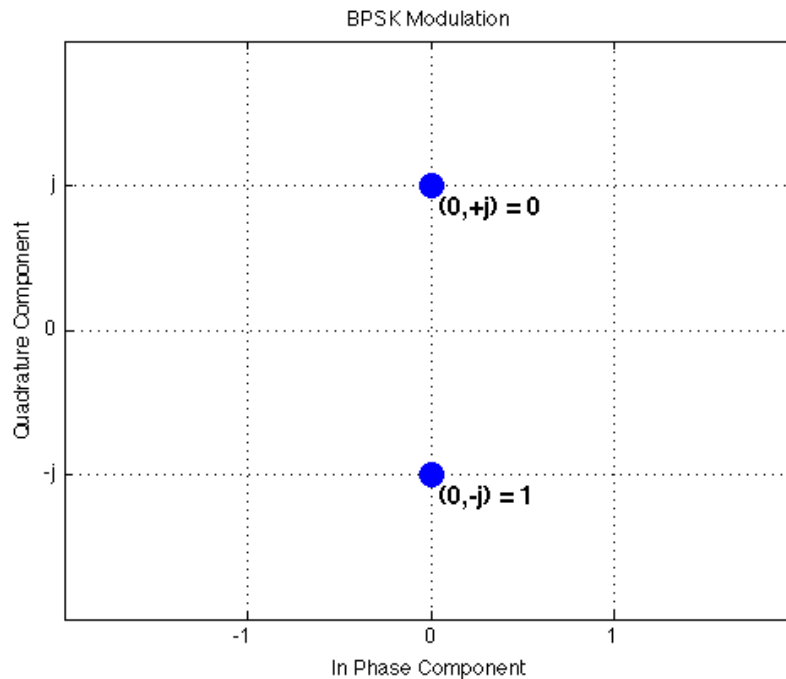


Figure 3: BPSK Symbol Map

set of received symbol samples, and corrects for a phase & frequency offset which equalizes all of the received symbols "close" to transmitted symbols (+j or -j).

Take a minute or two to think about this equalizer. While this works fairly well for BPSK, there's one unfortunate side effect that you'll be responsible for fixing. **What happens if the phase & frequency offset of the received symbol stream is greater than 90° ?** It turns out that in this case, the equalizer will probably guess wrong, i.e. it will assume that 1's are 0's and 0's are 1's. The next block will attempt to correct this issue.

(4) Sync/Preamble Detection & Binary Channel Correction

In this block, we're going to kill two birds with one stone: Check the equalizer's phase/frequency guess and fix it (if it's wrong), and detect the start of a valid data packet.

We can think of the output from the channel equalizer as a communication channel which beyond adding a small amount of noise, either flips **all** the transmitted symbols (when the equalizer wrongly estimates the phase offset, all the transmitted '+j's are received as '-j's and vice versa) or let's it through without any flipping (when the

equalizer correctly estimates the phase offset). At the receiver, we don't know whether the symbols were flipped or not - *yet*. This is where the 31 bits of preamble/sync come in handy. This signal, which we'll refer to as $p[n]$, is inserted at the beginning of each transmitted packet.

We will provide this 31-bit message, which has a particular form and some special properties. We'll come to these properties in a moment, but first, a short aside on correlation:

1.4.1 Correlation

We define the cross-correlation of two signals $f[n]$ and $g[n]$ as

$$(f \star g)[k] = \sum_{m=-\infty}^{\infty} f^*[m]g[k+m] \quad (1)$$

If $g[n] = f[n]$, then this is called the **autocorrelation** of the signal $f[n]$. The output of this equation is effectively how much the signal $f[n]$ looks like the signal $g[n]$ when g is offset from f by k samples.

Now, here are the special correlation properties of the provided PN preamble/sync $p[n]$

- First, $p[n]$ is what's called a pseudo-noise sequence, which means it doesn't *correlate* well with any other signal, i.e. $(p \star f)[k]$ is small for all k if $f[n] \neq p[n]$.
- Second, $p[n]$ only auto-correlates well with itself at zero-sample offset, i.e. $(p \star p)[0]$ is large, and $(p \star p)[k]$ is small for $k \neq 0$.

Can you guess how this might come in handy for the preamble detection? Here's what we're going to do: maintain a *sliding correlator*, which calculates the correlation between the last 31 symbols of the incoming signal $r[n]$ and the known sync/preamble signal $p[n]$. If we are receiving the beginning of a packet (i.e. the 31 bits of the sync/preamble field) and the polarity hasn't been flipped by the equalizer, the correlator will be large positive. If we are receiving the beginning of a packet and the polarity *has* been flipped, the correlator will be large negative!

Think about this for a moment!!!.

Let's work through an example to see how this works. Suppose that our preamble sequence is just 5 bits long (10110). This will be encoded to $p[n] = (-j, +j, -j, -j, +j)$ and will be attached to the beginning of every data packet at the transmitter (note here that the order of transmission is as shown, i.e. $-j$ is transmitted first followed by

$+j, -j, -j, +j$, eventually followed by the data) . Now let's say the received equalized symbol sequence in order of reception is:

$r[n] = (+j, -j, +j, -j, -j, -j, -j, +j, -j, -j, +j, \dots)$,i.e. $r[0] = +j$ is the first received symbol and $r[1] = -j$ is the second and so on.

Note that, this is just an example where all the equalized symbols exactly match up to the expected constellation points. In reality, however, the equalized symbol sequence will also include a small amount of noise. Carrying forward with our example, let's try and calculate the correlation of the received sequence with our expected preamble. The correlation of the first 5 received symbols with the preamble :

$$\begin{aligned} \text{Correlation} &= r^*[0]p[0] + r^*[1]p[1] + r^*[2]p[2] + r^*[3]p[3] + r^*[4]p[4] \\ &= (+j)^*(-j) + (-j)^*(+j) + (+j)^*(-j) + (-j)^*(-j) + (-j)^*(+j) \\ &= (-1) + (-1) + (-1) + (+1) + (-1) \\ &= -3 \end{aligned}$$

The correlation of the 2nd to 6th received symbols with the preamble :

$$\begin{aligned} \text{Correlation} &= r^*[1]p[0] + r^*[2]p[1] + r^*[3]p[2] + r^*[4]p[3] + r^*[5]p[4] \\ &= (-j)^*(-j) + (+j)^*(+j) + (-j)^*(-j) + (-j)^*(-j) + (-j)^*(+j) \\ &= (+1) + (+1) + (+1) + (+1) + (-1) \\ &= 3 \end{aligned}$$

and so on

Having looked at these calculations, try to convince yourselves of the following :

- In the absence of noise, the correlation value of a portion of the received sequence with a 5 bit long preamble will always lie between 5 and -5 .
- In the absence of noise, the correlation value of a portion of the received sequence with the preamble will always be a real number.
- The higher the correlation value (close to 5), the larger the possibility that we are looking at the start of a packet. In other words, the higher the value, the more closely the received sequence resembles the preamble.
- The higher the magnitude of the correlation value in the negative direction (close to -5), the larger the probability that we are looking at the start of a packet and the equalizer has made a mistake in the phase equalization. In other words, the higher the magnitude of correlation in the negative direction, the more closely the received sequence resembles a flipped version of the preamble. Note that if this is the case, we can be sure that the ensuing data is also flipped and can hence perform the necessary correction.

Hence, in the above received sequence, when we calculate the correlation of the 7th to 11th received symbols with the preamble :

$$\begin{aligned}
 \text{Correlation} &= r^*[6]p[0] + r^*[7]p[1] + r^*[8]p[2] + r^*[9]p[3] + r^*[10]p[4] \\
 &= (-j)^*(-j) + (+j)^*(+j) + (-j)^*(-j) + (-j)^*(-j) + (+j)^*(+j) \\
 &= (+1) + (+1) + (+1) + (+1) + (+1) \\
 &= 5
 \end{aligned}$$

We see a high positive value and can guess with a high confidence, that this is the start of the packet and that the received symbol number 12 onwards we should start receiving the actual transmitted message. Also we can guess with high confidence, that the equalizer **did not** make any mistake in the carrier and phase offset correction.

Note that this was just an example where 5 length preamble was used. In the lab, we will be using 31 bit preamble to minimize any risk of false detection. Hence, by a "high positive value" of correlation, we would mean something close to 31 and by "high negative value" of correlation, we would mean something close to -31.

When this correlation crosses some threshold $\pm C_{thresh}$, we will look for the peak of the correlation (to ensure that we've found the actual first symbol of the packet), and **make note of our guess for the starting bit of the packet preamble.** Additionally, we'll then **apply a binary correction (i.e. multiply the received symbols by -1) if the equalizer flipped our polarity.** Now the symbols are ready to be de-mapped into bits. You should **send the following 30+1 bytes (data + crc) to the symbol demapper.**

(5) Hard BPSK Demapper

Next, we'll implement a hard demapper to get bits back from the received symbols. For the 30+1 bytes of the packet, we'll demap each symbol into a bit. Here's how it works: Look at the BPSK symbol map shown in figure 3 above. If both symbols are sent with equal probability, we have a very simple decision rule for each symbol:

- If the received quadrature component $q[n] \geq 0$, then we guess that we've received a 0.
- If it is below zero, then we guess that we've received a 1.

Mathematically, we state this as

$$\hat{x}[n] = \begin{cases} 0 & q[n] \geq 0 \\ 1 & q[n] < 0 \end{cases} \quad (2)$$

where $\hat{x}[n]$ is our *estimate* of the transmitted symbol $x[n]$.

(6) Packet Decoder

We're almost there! Now we have a bitstream of estimates for each bit $\hat{x}[n]$. We now should separate the bitstream into two segments:

- the 30 bytes corresponding to the data
- the 1 byte corresponding to the crc

For a sanity check, we'll send the 30 bytes of message to an ASCII decoder and see what we've got. Finally, we'll send the 30 bytes of data plus the 1 byte CRC to a CRC error checker to see if there are any errors. Note that it's perfectly fine to decode the message whether or not the CRC is OK. If the CRC is wrong, you'll just get an entirely (or partially) garbled message.

(7) CRC Check

CRC is a set of bits (8 in our case) that are calculated and appended at the end of the data portion of the packet while transmitting. This allows for a verification of the correctness of the received message at the receiver. Note that it only detects errors and cannot correct them. For a detailed discussion on the concept of CRC, refer to the wikipedia article on the same (http://en.wikipedia.org/wiki/Cyclic_redundancy_check), especially go through the section titled "Computation of CRC". In this lab, we will be computing an 8-bit CRC and hence we would need a 9-bit CRC divisor. We choose the CRC divisor bit sequence to be:

1 0 0 0 0 1 1 1

This CRC is also known as CRC-8-CCITT.

2 Lab 4 Walkthrough

Open RX_lab4.vi. Most of the control traffic and the initialization has been taken care of. The subVI "generate system parameters" does the same function as it did in the transmitter code, i.e. it outputs a cluster named "PSK system parameters" which has 2 entries:

- A numeric "Samples per symbol"
- An array of complex values "Symbol map"

Note that, in this lab we would be using BPSK modulation and the corresponding constellation would look like 3.

The next step is the subVI "validate and generate filter" which generates a "matched filter" which is designed to in order to undo the effects of the Root Raised Cosine Filter applied on the transmitter side.

The general procedure of receiving data from the USRPs is to allocate a buffer of some size and pass it to the USRP driver which then fills it with the received samples and passes it back. This takes place in the subVI "niUSRP Fetch Rx Data (poly).vi" inside the while loop. The size of this buffer is decided by the input "Acq Duration [sec]" which is set ot "50 ms" by default. This implies, the buffer is so allocated that whenever the USRP is accessed, we get "50 ms " worth of received data. To achieve this, the buffer size is such that it can hold $AcqDuration * RXSamplingRate$ number of complex samples.

A resampled version of these received samples alongwith the "PSK system parameters" and the "matched filter" are passed on to the subVI "Demodulate PSK". This is where the equalization takes place. As described earlier, the equalization has been taken care of. Look at it as a block which removes the effects of the transmit side filter and equalizes any carrier frequency or phase offsets and outputs a set of equalized samples (you will receive "Samples per symbol" number of samples for every BPSK symbol).

The output of "Demodulate PSK" is then truncated and passed on to the subVI that plots the received constellation diagram. In order to avoid extensive data processing in each iteration of the while loop (each time a set of samples is received), we decided to push all the processing outside the while loop. This implies, whenever RX_lab4.vi is run, a large set of samples is acquired over and over again, however, only the equalization is taken care of and the received samples are plotted on the constellation diagram. Eventually, when you hit "STOP", the code stops acquiring received samples (exits the while loop) and starts processing the last batch of received samples. The first step of this data processing is the decimation. The complex waveform is unbundled and the array of complex samples are passed on to the subVI "Decimate.vi". This is where your task starts.

(1) Decimate

Open "Decimate.vi", the input "Signal In" is an array of equalized complex samples, you will have "Samples per symbol" number of complex samples per BPSK symbol. Another input is "Samples per symbol". **Your task is to decimate the "Signal In" array which means to discard the extra samples and form a stream of BPSK symbols.** This is in some sense, the converse of the upsampling operation you performed on the transmitter. You are guaranteed that the first sample corresponds to a BPSK symbol and from thereon, you need to skip "Samples per symbol" - 1 samples to get to the next BPSK symbol and so on, till the end of the array.

A typical input and output scenario is shown in 4 for verification purposes.

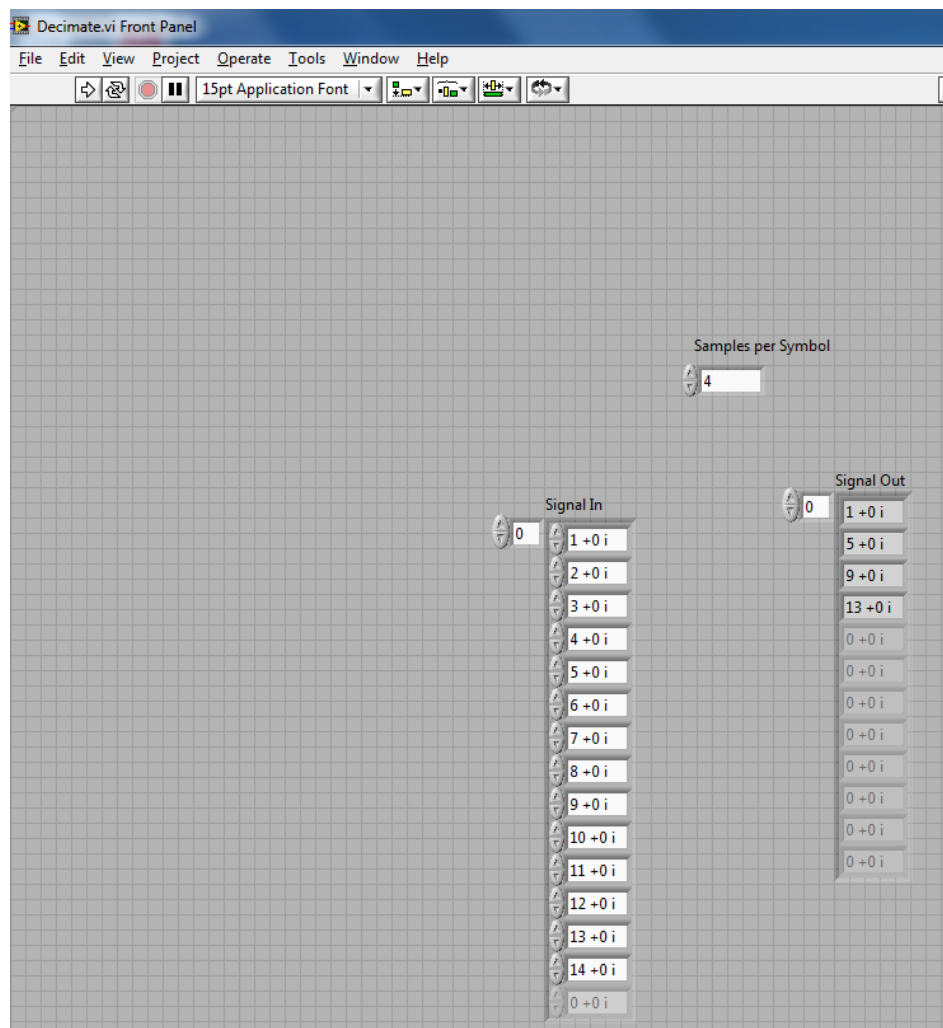


Figure 4: Output of Decimate.vi

(2) Preamble Detector & Channel Correcter

This stream of BPSK symbols is then passed on to the subVI "preamble_detector_channel_corrector.vi". This subVI, as the name suggests should check for the start of the preamble, corrects the channel in case the equalizer flipped polarities, and then outputs the corrected symbols corresponding to the 30 + 1 bytes of data + crc. **Finish this subVI.** The subVI "preamble_generator" will output the 31 preamble bits for you to compare against. While implementing the preamble detector and channel corrector, keep in mind the following:

- preamble_generator generates the bits corresponding to the preamble, and you should ensure that these bits are mapped according to the BPSK constellation in 3 before comparing it with the received symbols.
- The input array "Input Symbols" which consists of the received symbols is stored in the order of reception (i.e. Input Symbols[0] is the first received symbol) and the "preamble_generator" outputs the preamble bits in order of transmission (i.e. preamble[0] is the first preamble bit transmitted), thus while calculating the correlations, you will not need to flip any of these arrays.
- Notice the correlation formulae describe in the primer carefully, you will need to form the complex conjugate of one of the arrays before multiplying it with the other.
- The correlation value in general would be a complex number. However, as our calculations in the primer showed, if there is no noise, then the correlation value should turn out to be a real number. Even in the general case of noisy symbols, it is fair to discard the imaginary part of the correlation value and only consider the real part while comparing it with the threshold $\pm C_{thresh}$.
- As noticed in the primer, for a 31 bit long preamble, the correlation value should be close to 31 to confidently say "This is the start of the packet" and it should be close to -31 to confidently say "This is the start of the packet and the equalizer made a mistake while correcting phase offset". A typical value of the threshold can be chosen to be 30, i.e. compare the magnitude of the correlation with 30 to check for start of packet and see its sign to decide whether or not to correct the symbols.
- The correction (if necessary) only involves flipping the polarity of the received complex samples.
- Eventually, output the corrected version of the complex samples that correspond to the data + CRC portion of the packet. That is, discard the preamble and

output the succeeding $(30 + 1) * 8$ BPSK symbols which correspond to the data + CRC portion of the packet.

(3) Packet Decoder

This array of "Corrected data+CRC" is passed on to the subVI "packet_decoder.vi". In this subVI, you are supposed to do the following:

- Make a hard decision on the received symbols and map each one of them to a bit. **Complete the subVI "Hard Demapper.vi" which performs this hard demapping.** Refer to the primer's section on Hard demapping for more information.
- Each byte of the data portion corresponds to an ASCII character, hence you should have 30 characters (corresponding to the 30 bytes) in the message. **Convert this stream of bits to an array of 30 integers corresponding to the ASCII values.** You can use the subVI "binary_to_decimal" for this purpose.
- Eventually, just to make sure that the received message is the correct one, you should perform a CRC check. **Complete the subVI "CRC_check.vi" which inputs the CRC appended message and the CRC polynomial and outputs whether or not the check was succesful.** For more information on CRC checking, refer to the primer.

Eventually, this ASCII array will be converted to text and displayed on the front panel of RX_lab4.vi.

3 Verification

In order to verify "Decimate.vi", try to replicate 4. For the overall verification, do the following:

- Start the TX_lab4.vi with the correct parameters. The parameters that need to be filled are similar to the previous lab. However, this time around, you do not have the option of sending a QPSK stream. The indicator "data + CRC length" shows the number of bits in the data + CRC portion of the packet. For a sanity check, confirm that this indicator shows "248" which corresponds to $(30+1)*8$ bits in the data + CRC portion of the packet.

- Start the RX_lab4.vi with the correct parameters. The parameters that need to be filled are similar to the previous lab, however, the extra parameter to be filled is "data length". Since, we are sending a 30 byte data, this control should be set to "240" which corresponds to the number of bits in the data portion of the packet.
- As described in the walkthrough, the receiver starts collecting samples, equalizing and plotting them. No data processing is being performed yet. Hit the "STOP" button, to stop collecting samples, this automatically initiates the data processing chain (preamble detection and channel correction and packet decoding) on the last set of received symbols.
- If you receive a meaningful message on the "Output Message" indicator and a green light on the "CRC check" indicator, you are Done!!!